Data Engineering

Lifestyle Factors and Their Impact on Students

By:

اميرة محسن الغموي 444005124

أمجاد احمد هوساوي 444003283

نورة محمد الكبكبي 444003676

وئام راشد 444015412

Project Report: Student Lifestyle Dashboard

Design Overview

The Student Lifestyle Dashboard is a comprehensive data processing and visualization pipeline designed to analyze and present student-related data, focusing on academic performance, daily activities, and stress levels. The system is structured into four distinct phases, each leveraging different databases and processing technologies to handle data efficiently and provide actionable insights. The pipeline is implemented in Python and uses Streamlit for the front-end dashboard, ensuring an interactive and user-friendly interface.

Phase 1: Relational Database (SQLite)

This phase involves loading a CSV dataset (student_lifestyle_dataset.csv) into a SQLite database. The data is normalized into four tables: Genders, Students, Grades, and Daily_Activities. Key operations include:

- **Data Loading**: The CSV is read into a Pandas DataFrame and inserted into SQLite tables.
- **Schema Design**: Tables with appropriate primary and foreign key constraints are created to ensure data integrity.
- **CRUD Operations**: Basic Create, Read, Update, and Delete operations are implemented and tested.
- **Query Optimization**: Indexes (e.g., on study_hours) and optimized JOIN queries are used to improve performance, with execution times measured to demonstrate efficiency.

```
# Genders
cursor.execute("""
CREATE TABLE Genders (
    Gender_ID INTEGER PRIMARY KEY AUTOINCREMENT,
    Gender_Name TEXT UNIQUE
)
""")

# Students
cursor.execute("""
CREATE TABLE Students (
    Student_ID INTEGER PRIMARY KEY,
    Gender_ID INTEGER,
    Stress_Level TEXT,
    FOREIGN KEY (Gender_ID) REFERENCES Genders(Gender_ID)
)
""")

# Grades
cursor.execute("""
CREATE TABLE Grades (
    Grade_ID INTEGER PRIMARY KEY AUTOINCREMENT,
    Student_ID INTEGER,
    Grade_Value REAL,
    FOREIGN KEY (Student_ID) REFERENCES Students(Student_ID)
)
""")

# Daily Activities
cursor.execute("""
CREATE TABLE Daily_Activities (
    Activity_ID INTEGER PRIMARY KEY AUTOINCREMENT,
    Student_ID INTEGER,
    Study_Hours REAL,
    Extracurricular_Hours REAL,
    Sleep_Hours REAL,
    Social_Hours REAL,
    Physical_Activity_Hours REAL,
    FOREIGN KEY (Student_ID) REFERENCES Students(Student_ID)
)
```

```
#----------------CREATE
print()

cursor.execute("DELETE FROM Students WHERE Student_ID = 2001")
conn.commit()
cursor.execute("INSERT INTO Students (Student_ID, Gender_ID, Stress_Level) VALUES (2001, 2, 'Medium')")
conn.commit()
cursor.execute("SELECT * FROM Students WHERE Student_ID = 2001 ")
print("Create row 2001 in the last of student table")
print(cursor.fetchall())


#----------------DELETE
print()
cursor.execute("DELETE FROM Students WHERE Student_ID = 2001")
conn.commit()
cursor.execute("SELECT * FROM Students WHERE Student_ID = 2001")
print("Delete row 2001 from the student")
print(cursor.fetchall())
```

```
The row 1 in student
[(1, 1, 'Medium')]

Update the row 1 in student
[(1, 1, 'Medium')]

Create row 2001 in the last of student table
[(2001, 2, 'Medium')]

Delete row 2001 from the student
[]
```

```
Query Optimization BY INDEXING
Time without index: 0.0008745193481445312
Time with index:    0.0004086494445800781
```

Phase 2: NoSQL Database (TinyDB)

In this phase, the same dataset is ingested into TinyDB, a lightweight NoSQL database. The focus is on flexible, document-based storage:

- **Data Insertion**: The dataset is converted into JSON-like documents, each representing a student's profile, including grades, stress levels, and daily activities.
- **Queries**: Queries identify students eligible for scholarships (GPA ≥ 7), counseling (moderate/high stress), and sports scholarships (physical activity ≥ 5 hours/day). Low-performing students (GPA ≤ 2) are removed.
- **Advantages**: TinyDB's simplicity allows rapid prototyping and flexible querying, suitable for small-scale, dynamic datasets.

```python
def phase2_tinydb(df: pd.DataFrame, tiny_path: Path) -> None:
    print(banner("Phase 2 – NoSQL (TinyDB)"))
    db = TinyDB(tiny_path)
    db.truncate()
    sample = df
    docs = [{
        "student_id": int(r.Student_ID),
        "study_hours_per_day": float(r.Study_Hours_Per_Day),
        "extracurricular_hours_per_day": float(r.Extracurricular_Hours_Per_Day),
        "sleep_hours_per_day": float(r.Sleep_Hours_Per_Day),
        "social_hours_per_day": float(r.Social_Hours_Per_Day),
        "physical_activity_hours_per_day": float(r.Physical_Activity_Hours_Per_Day),
        "stress_level": r.Stress_Level,
        "gender": r.Gender,
        "grades": float(r.Grades),
    } for r in sample.itertuples(index=False)]
    db.insert_multiple(docs)
    print("Inserted 100 student records!")
    # Students with GPA >= 7 for scholarships
    Student = Query()
    print("\nStudents eligible for scholarships (GPA >= 7):")
    scholarship_students = db.search(Student.grades >= 7)
    for student in scholarship_students:
        print(f"Student ID: {student['student_id']}, GPA: {student['grades']}")
    # Students with Moderate or High stress level for counseling
    print("\nStudents recommended for counseling (Moderate or High stress):")
    counseling_students = db.search(Student.stress_level.one_of(['Moderate', 'High']))
    for student in counseling_students:
        print(f"Student ID: {student['student_id']}, Stress Level: {student['stress_level']}")
    # Students with high physical activity (>= 5 hours/day) for sports scholarships
    print("\nStudents eligible for sports scholarships (Physical Activity >= 5 hours/day):")
    sports_students = db.search(Student.physical_activity_hours_per_day >= 5)
    for student in sports_students:
        print(f"Student ID: {student['student_id']}, Physical Activity: {student['physical_activity_hours_per_day']} hours")
    # Delete students with GPA <= 2
    db.remove(Student.grades <= 2)
    print("\nDeleted students with GPA <= 2!")
    # Step 10: Count Documents
    print("\nTotal number of students:", len(db))
```

Phase 3: Stream Processing (PySpark)

This phase uses PySpark for distributed data processing, simulating a streaming pipeline:

- **Data Processing**: The dataset is processed to filter high-performing students (grades ≥ 9.75), select specific columns, and sort by grades.
- **Output**: Results are saved as CSV files in three directories (output_students, output_students2, output_students3).

- **Scalability**: PySpark ensures the pipeline can handle larger datasets in a distributed environment, with local execution for testing.

```
!pip install -q pyspark
```

```
#Start SparkSession
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("StudentsLifeStyle").getOrCreate()

df = spark.read.csv("/content/student_lifestyle_cleaned.csv", header=True )

# Create DataFrame

df.show()
```

```
+----------+------------------+--------------------------+------------------+-------------------+
|student_id|study_hours_per_day|extracurricular_hours_per_day|sleep_hours_per_day|social_hours_per_day|
+----------+------------------+--------------------------+------------------+-------------------+
|         1|               6.9|                       3.8|               8.7|                2.8|
|         2|               5.3|                       3.5|               8.0|                4.2|
|         3|               5.1|                       3.9|               9.2|                1.2|
|         4|               6.5|                       2.1|               7.2|                1.7|
```

```
# Filter students with grades above 9.75/10
filtered_df = df.filter(df["grades"] >= 9.75)
filtered_df.show()
```

Python

```
-------+-----------------------------+------------+------+------+
er_day|physical_activity_hours_per_day|stress_level|gender|grades|
-------+-----------------------------+------------+------+------+
    3.1|                          0.8|        High|Female|  10.0|
    5.3|                          3.1|        High|  Male|   9.8|
    3.3|                          3.7|        High|  Male|  9.82|
    2.5|                          0.8|        High|  Male|  9.78|
    0.1|                          2.0|        High|  Male|  9.75|
-------+-----------------------------+------------+------+------+
```

```python
selected_df = df.select(df["student_id"],df["grades"])
selected_df.show()
```

Python

```
+----------+------+
|student_id|grades|
+----------+------+
|         1|  7.48|
|         2|  6.88|
|         3|  6.68|
|         4|   7.2|
|         5|  8.78|
|         6|  7.12|
```

```python
sorted_df = df.orderBy(df["grades"].desc())
sorted_df.show()
```

Python

```
-------+------------------------------+------------+------+------+
per_day|physical_activity_hours_per_day|stress_level|gender|grades|
-------+------------------------------+------------+------+------+
    3.3|                           3.7|        High|  Male|  9.82|
    5.3|                           3.1|        High|  Male|   9.8|
    2.5|                           0.8|        High|  Male|  9.78|
    0.1|                           2.0|        High|  Male|  9.75|
    2.1|                           2.1|        High|  Male|  9.68|
    0.6|                           2.0|        High|  Male|  9.68|
    3.0|                           3.6|        High|Female|  9.65|
    1.9|                           1.7|        High|  Male|  9.62|
    5.7|                           0.7|        High|Female|   9.6|
```

```python
def phase3_spark(clean_csv: Path, out_dir: Path) -> list[Path]:
    print(banner("Phase 3 – Stream Processing (PySpark)"))

    spark = SparkSession.builder.appName("StudentsLifeStyle").master("local[*]").getOrCreate()
    sdf = spark.read.csv(str(clean_csv), header=True, inferSchema=True)

    sdf = sdf.withColumnRenamed("Grades", "grades")
    filtered = sdf.filter(sdf["grades"] >= 9.75)
    selected = sdf.select("student_id", "grades")
    sorted_df = sdf.orderBy(sdf["grades"].desc())

    dirs = [
        out_dir / "output_students",
        out_dir / "output_students2",
        out_dir / "output_students3",
    ]
    filtered.write.mode("overwrite").option("header", "true").csv(str(dirs[0]))
    selected.write.mode("overwrite").option("header", "true").csv(str(dirs[1]))
    sorted_df.write.mode("overwrite").option("header", "true").csv(str(dirs[2]))

    spark.stop()
    print("Spark outputs →", ", ".join(d.name for d in dirs))
    return dirs
```

Phase 4: Integration and Reporting

This phase consolidates data from all previous phases:

- **Data Integration**: SQLite, TinyDB, and PySpark outputs are merged into a single Pandas DataFrame, handling schema differences and ensuring consistency.
- **Output**: A combined CSV (combined_dataset.csv) is generated, including a top_performer flag for high-achieving students.
- **Reporting**: Aggregate insights, such as average grades by stress level, are computed and displayed.

```python
def phase4_integration(db_path: Path, tiny_path: Path, spark_dirs: list[Path], out_csv: Path) -> pd.DataFrame:
    print(banner("Phase 4 – Integration & Reporting"))

    # Read phase 1 data
    with sqlite3.connect(db_path) as conn:
        students = pd.read_sql("SELECT * FROM Students", conn)
        grades   = pd.read_sql("SELECT * FROM Grades", conn)
        acts     = pd.read_sql("SELECT * FROM Daily_Activities", conn)

    # Read phase 2 data
    nosql_df = pd.DataFrame(TinyDB(tiny_path).all())

    # Read phase 3 data
    spark_frames: dict[str, pd.DataFrame] = {}
    for d in spark_dirs:
        files = sorted(glob.glob(str(d / "*.csv")))
        if files:
            spark_frames[d.name] = pd.read_csv(files[0])

    relational = (
        students
        .merge(grades, on="student_id", how="left")
        .merge(acts,   on="student_id", how="left")
    )
    relational.columns = map(str.lower, relational.columns)
    nosql_df["student_id"] = nosql_df["student_id"].astype(int)

    combined = relational.merge(nosql_df, on="student_id", how="outer", suffixes=("", "_nosql"))

    if "output_students" in spark_frames:
        perf = spark_frames["output_students"][["student_id"]].astype(int)
        perf["top_performer"] = True
        combined = combined.merge(perf, on="student_id", how="left")
        combined["top_performer"].fillna(False, inplace=True)

    combined.to_csv(out_csv, index=False)
    print(f"Combined CSV saved → {out_csv.name} ({len(combined):,} rows)")

    print(banner("Average Grade by Stress Level"))
    if "grade_value" in combined.columns:
        print(combined.groupby("stress_level")["grade_value"].mean().round(2))

    return combined
```

Dashboard (Streamlit)

The Streamlit dashboard visualizes the outputs of all phases:

- **Components**: Displays tables for SQLite data, TinyDB documents, PySpark outputs, and the combined dataset.
- **Visualizations**: Includes a bar chart showing average grades by stress level.
- **Interactivity**: Users can expand instructions and explore data dynamically, with a refresh option to update after pipeline reruns.

The pipeline is executed via streamlit run main.py, ensuring seamless integration of data processing and visualization.

```python
def launch_dashboard(db_path: Path, tiny_path: Path, spark_dirs: list[Path], combined_csv: Path) -> None:
    # ---- Phase 1 (SQLite)
    conn = sqlite3.connect(db_path)
    students_df = pd.read_sql_query("SELECT * FROM Students", conn)
    conn.close()

    # ---- Phase 2 (TinyDB)
    records = TinyDB(tiny_path).all()
    tiny_df = pd.DataFrame(records)

    # ---- Phase 3 (Spark outputs)
    streamed = []
    for d in spark_dirs:
        for f in sorted(Path(d).glob("*.csv")):
            streamed.append(pd.read_csv(f))
    stream_df = pd.concat(streamed, ignore_index=True) if streamed else pd.DataFrame()
    # ---- Phase 4 (Combined CSV)
    combined = pd.read_csv(combined_csv)
    # ---- UI ------------------------------------------------------------
    st.title("🎓 Student Data Dashboard")

    with st.expander("ℹ️ Instructions", expanded=False):
        st.markdown("This dashboard illustrates the outputs of all four pipeline phases …")

    st.subheader("📋 Phase 1: Students (Relational DB)")
    st.dataframe(students_df, use_container_width=True)

    st.subheader("🗃️ Phase 2: Lifestyle Documents (TinyDB)")
    st.dataframe(tiny_df, use_container_width=True)

    st.subheader("📡 Phase 3: Streamed Grades (Spark CSVs)")
    if not stream_df.empty:
        st.dataframe(stream_df, use_container_width=True)
    else:
        st.info("No stream data found – run Phase 3 first.")

    st.subheader("🌸 Phase 4: Combined Dataset")
    st.dataframe(combined, use_container_width=True)

    st.subheader("📊 Aggregate Insights")
    if "grades" in tiny_df.columns:
        st.bar_chart(tiny_df.groupby("stress_level")["grades"].mean())

    st.success("Dashboard ready! Reload after rerunning the pipeline to refresh data.")
```

Deploy

🎓 Student Data Dashboard

ℹ Instructions

📋 Phase 1: Students (Relational DB)

	student_id	gender_id	stress_level
0	1	1	Medium
1	2	2	Low
2	3	1	Low
3	4	1	Moderate
4	5	1	High
5	6	2	Moderate
6	7	1	High
7	8	1	High
8	9	1	Low
9	10	2	Moderate

🗃 Phase 2: Lifestyle Documents (TinyDB)

	student_id	study_hours_per_day	extracurricular_hours_per_day	sleep_hours_per_day	social_hours_per_day	physical_activity_hours_per_day	stress_level	gender	grades
0	1	6.9	3.8	8.7	2.8	1.8	Moderate	Male	7.48
1	2	5.3	3.5	8	4.2	3	Low	Female	6.88
2	3	5.1	3.9	9.2	1.2	4.6	Low	Male	6.68
3	4	6.5	2.1	7.2	1.7	6.5	Moderate	Male	7.2
4	5	8.1	0.6	6.5	2.2	6.6	High	Male	8.78
5	6	6	2.1	8	0.3	7.6	Moderate	Female	7.12
6	7	8	0.7	5.3	5.7	4.3	High	Male	7.7
7	8	8.4	1.8	5.6	3	5.2	High	Male	8
8	9	5.2	3.6	6.3	4	4.9	Low	Male	7.05
9	10	7.7	0.7	9.8	4.5	1.3	Moderate	Female	6.9

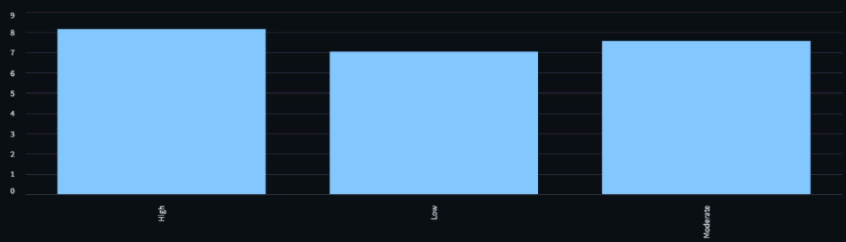
📡 Phase 3: Streamed Grades (Spark CSVs)

	student_id	study_hours_per_day	extracurricular_hours_per_day	sleep_hours_per_day	social_hours_per_day	physical_activity_hours_per_day	stress_level	gender	grades	gen
0	52	9	2.6	8.5	3.1	0.8	High	Female	10	
1	871	9.7	0.1	5.8	5.3	3.1	High	Male	9.8	
2	1230	9.8	1.9	5.3	3.3	3.7	High	Male	9.82	
3	1455	9.4	1.8	9.5	2.5	0.8	High	Male	9.78	
4	1589	9.8	3	9.1	0.1	2	High	Male	9.75	
5	52	9	2.6	8.5	3.1	0.8	High	Female	10	
6	871	9.7	0.1	5.8	5.3	3.1	High	Male	9.8	
7	1230	9.8	1.9	5.3	3.3	3.7	High	Male	9.82	
8	1455	9.4	1.8	9.5	2.5	0.8	High	Male	9.78	
9	1589	9.8	3	9.1	0.1	2	High	Male	9.75	

🌸 Phase 4: Combined Dataset

	student_id	gender_id	stress_level	grade_id	grade_value	activity_id	study_hours	extracurricular_hours	sleep_hours	social_hours	physical_activity_hours	study_hours_per_
0	1	1	Medium	1	7.48	1	6.9	3.8	8.7	2.8	1.8	
1	2	2	Low	2	6.88	2	5.3	3.5	8	4.2	3	
2	3	1	Low	3	6.68	3	5.1	3.9	9.2	1.2	4.6	
3	4	1	Moderate	4	7.2	4	6.5	2.1	7.2	1.7	6.5	
4	5	1	High	5	8.78	5	8.1	0.6	6.5	2.2	6.6	
5	6	2	Moderate	6	7.12	6	6	2.1	8	0.3	7.6	
6	7	1	High	7	7.7	7	8	0.7	5.3	5.7	4.3	
7	8	1	High	8	8	8	8.4	1.8	5.6	3	5.2	
8	9	1	Low	9	7.05	9	5.2	3.6	6.3	4	4.9	
9	10	2	Moderate	10	6.9	10	7.7	0.7	9.8	4.5	1.3	

📊 Aggregate Insights



Dashboard ready! Reload after rerunning the pipeline to refresh data.

Sample Outputs

```

Phase 1 – Relational DB (SQLite)
--------------------------------
Loaded 2,000 rows from student_lifestyle_dataset.csv

CRUD Operations
----------------------------------------
The row 1 in student:
[(1, 1, 'Moderate')]
Updated row 1 in student:
[(1, 1, 'Medium')]
Created row 2001 in student table:
[(2001, 2, 'Medium')]
Deleted row 2001 from the student:
[]

```

main.py | output.txt

output.txt

```
Query Optimization Using JOIN Queries
      student_id  study_hours stress_level
0              3          5.1          Low
1             47          5.1          Low
2             59          5.1          Low
3             63          5.1         High
4            152          5.1          Low
...          ...          ...          ...
1975        1106         10.0         High
1976        1287         10.0         High
1977        1289         10.0         High
1978        1469         10.0         High
1979        1612         10.0         High

[1980 rows x 3 columns]
Time to process: 0.004999399185180664

Query Optimization Using Aggregation GROUP BY
      student_id gender_name stress_level  Total_Study_Hours  Average_Grade
0            101        Male         High               10.0           8.52
1            279        Male         High               10.0           8.90
2            308      Female         High               10.0           8.15
3            312        Male         High               10.0           8.60
4            364      Female         High               10.0           8.57
...          ...         ...          ...                ...            ...
1995        1693      Female          Low                5.0           7.22
1996        1704        Male          Low                5.0           6.60
1997        1799        Male         High                5.0           6.58
1998        1858        Male          Low                5.0           6.18
1999        1912      Female          Low                5.0           7.45

[2000 rows x 5 columns]
Time to process: 0.007122516632080078
```

```
Time to process: 0.007122516632080078
✔  SQLite populated and tested → student_lifestyle.db

Phase 2 – NoSQL (TinyDB)
------------------------
Inserted 100 student records!

Students eligible for scholarships (GPA >= 7):
Student ID: 1, GPA: 7.48
Student ID: 4, GPA: 7.2
Student ID: 5, GPA: 8.78
Student ID: 6, GPA: 7.12
Student ID: 7, GPA: 7.7
Student ID: 8, GPA: 8.0
Student ID: 9, GPA: 7.05
Student ID: 11, GPA: 8.57
Student ID: 12, GPA: 7.42
Student ID: 13, GPA: 7.05
Student ID: 14, GPA: 7.18
Student ID: 15, GPA: 8.5
Student ID: 16, GPA: 8.0
Student ID: 17, GPA: 8.4
```

main.py | output.txt

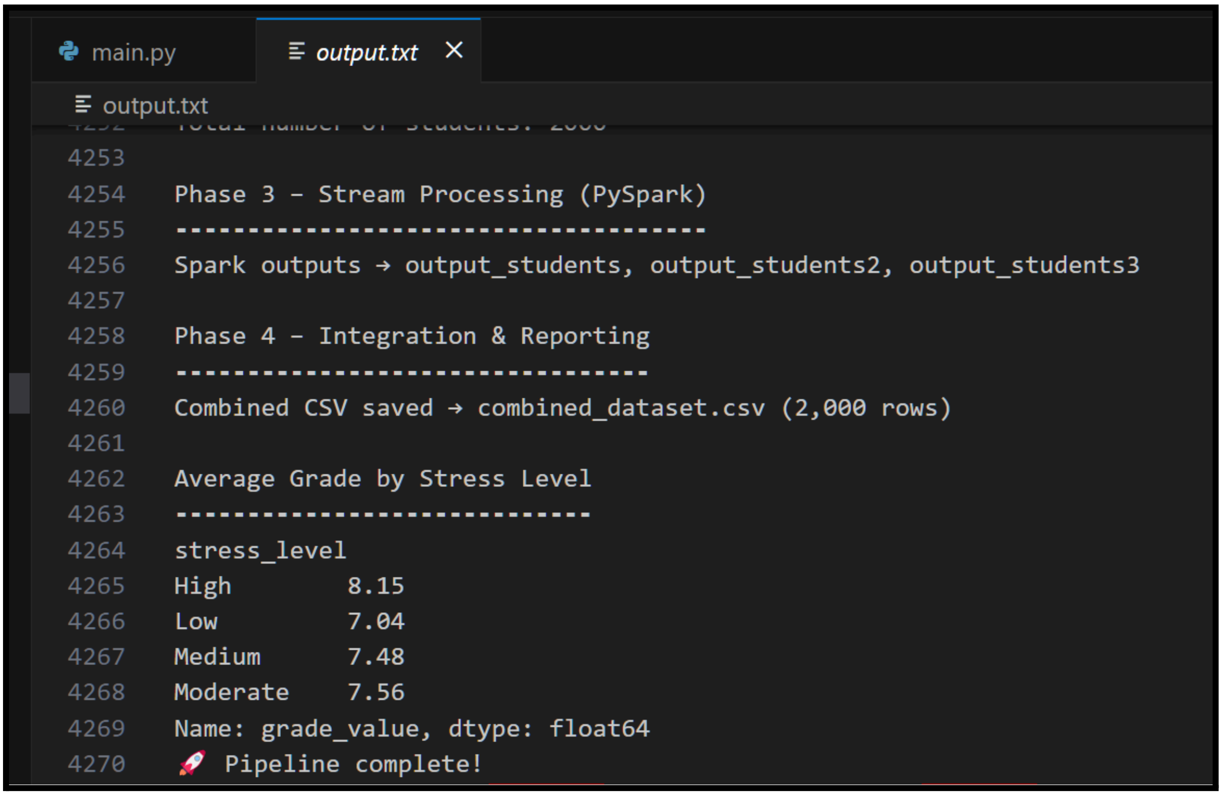
output.txt

```
Students recommended for counseling (Moderate or High stress):
Student ID: 1, Stress Level: Moderate
Student ID: 4, Stress Level: Moderate
Student ID: 5, Stress Level: High
Student ID: 6, Stress Level: Moderate
Student ID: 7, Stress Level: High
Student ID: 8, Stress Level: High
Student ID: 10, Stress Level: Moderate
Student ID: 11, Stress Level: High
Student ID: 12, Stress Level: Moderate
Student ID: 13, Stress Level: High
Student ID: 15, Stress Level: High
Student ID: 16, Stress Level: Moderate
Student ID: 17, Stress Level: High
Student ID: 18, Stress Level: High
Student ID: 19, Stress Level: High
Student ID: 20, Stress Level: Moderate
```

main.py | output.txt

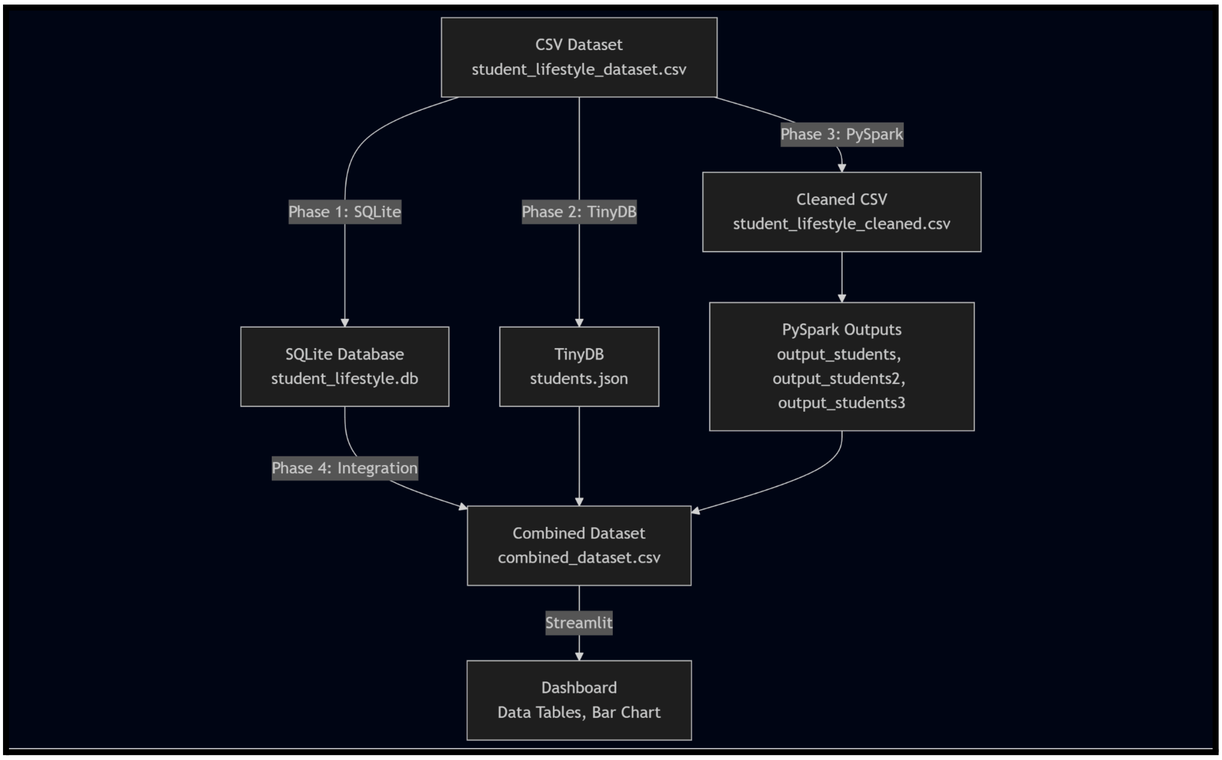
output.txt

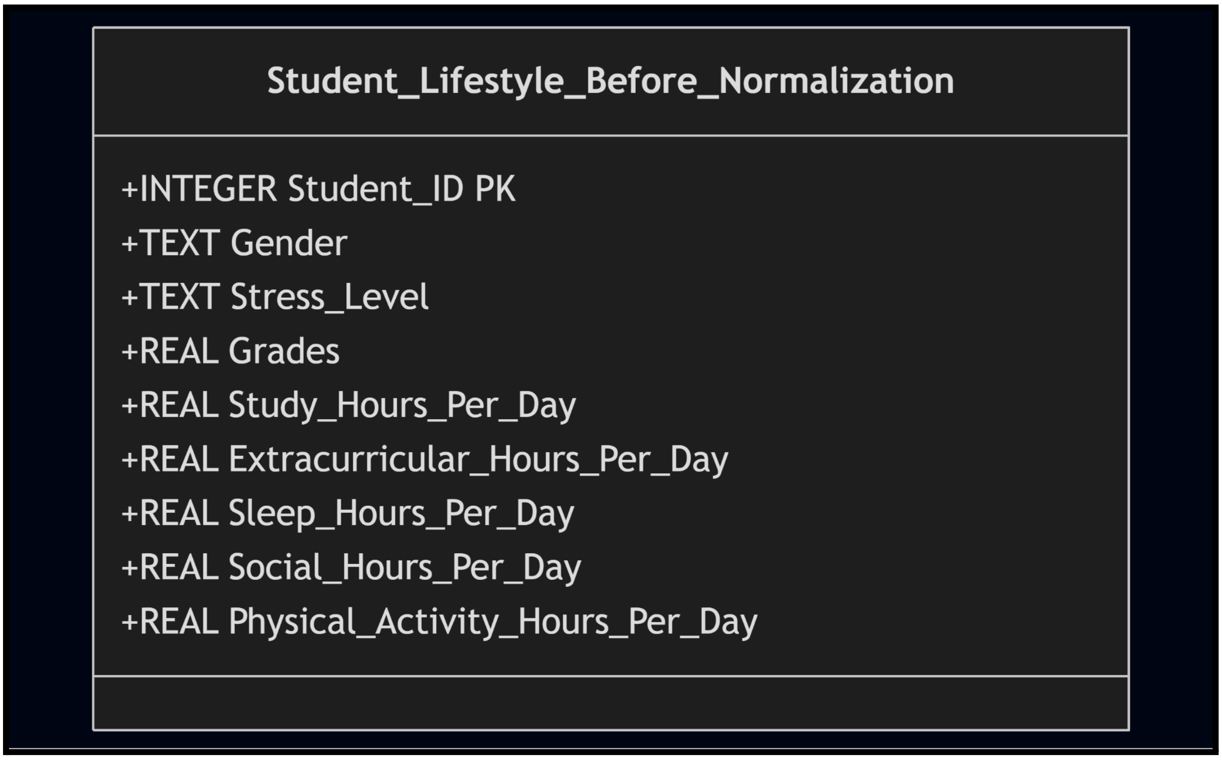
```
Students eligible for sports scholarships (Physical Activity >= 5 hours/day):
Student ID: 4, Physical Activity: 6.5 hours
Student ID: 5, Physical Activity: 6.6 hours
Student ID: 6, Physical Activity: 7.6 hours
Student ID: 8, Physical Activity: 5.2 hours
Student ID: 15, Physical Activity: 7.3 hours
Student ID: 16, Physical Activity: 8.4 hours
Student ID: 19, Physical Activity: 9.0 hours
Student ID: 21, Physical Activity: 7.3 hours
Student ID: 25, Physical Activity: 8.6 hours
Student ID: 28, Physical Activity: 6.5 hours
Student ID: 29, Physical Activity: 6.7 hours
Student ID: 32, Physical Activity: 7.5 hours
Student ID: 33, Physical Activity: 7.9 hours
Student ID: 35, Physical Activity: 6.3 hours
Student ID: 40, Physical Activity: 6.5 hours
Student ID: 43, Physical Activity: 9.2 hours
Student ID: 53, Physical Activity: 5.6 hours
Student ID: 55, Physical Activity: 7.4 hours
```

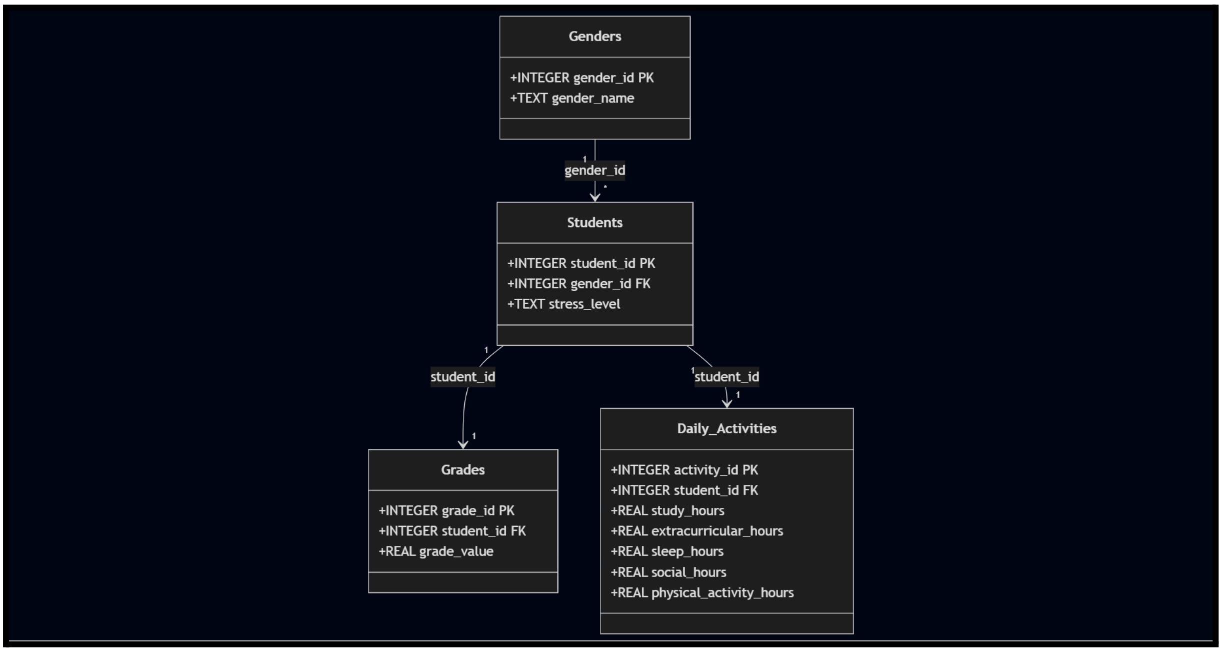


```
main.py    output.txt
output.txt
Total number of students: 2000

Phase 3 – Stream Processing (PySpark)
-------------------------------------
Spark outputs → output_students, output_students2, output_students3

Phase 4 – Integration & Reporting
---------------------------------
Combined CSV saved → combined_dataset.csv (2,000 rows)

Average Grade by Stress Level
-----------------------------
stress_level
High        8.15
Low         7.04
Medium      7.48
Moderate    7.56
Name: grade_value, dtype: float64
🚀 Pipeline complete!
```

Diagrams




Student_Lifestyle_Before_Normalization
+INTEGER Student_ID PK
+TEXT Gender
+TEXT Stress_Level
+REAL Grades
+REAL Study_Hours_Per_Day
+REAL Extracurricular_Hours_Per_Day
+REAL Sleep_Hours_Per_Day
+REAL Social_Hours_Per_Day
+REAL Physical_Activity_Hours_Per_Day


Genders
+INTEGER gender_id PK
+TEXT gender_name
gender_id
Students
+INTEGER student_id PK
+INTEGER gender_id FK
+TEXT stress_level
student_id
student_id
Grades
+INTEGER grade_id PK
+INTEGER student_id FK
+REAL grade_value
Daily_Activities
+INTEGER activity_id PK
+INTEGER student_id FK
+REAL study_hours
+REAL extracurricular_hours
+REAL sleep_hours
+REAL social_hours
+REAL physical_activity_hours

Lessons Learned

1. **Database Trade-offs**:

   SQLite: Well-suited for structured, relational data with strong consistency; however, it is less flexible for dynamic schemas. Indexing significantly enhanced query performance (e.g., accelerated study_hours queries).

   TinyDB: Facilitated simplified NoSQL prototyping but lacked scalability for extensive datasets or complex queries.

   PySpark: Effective for distributed processing, though it proved excessive for small datasets, resulting in overhead in local
   configurations.

2. **Data Integration Challenges**:

   Integrating data from diverse sources (SQL, NoSQL, Spark) necessitated meticulous schema alignment and management of missing values. Utilizing Pandas for integration was beneficial, but memory-intensive for larger datasets.

3. **Query Optimization**:

   Implementing indexing in SQLite significantly decreased query times, highlighting the critical importance of database optimization.

   JOIN and GROUP BY queries in SQLite yielded valuable insights but required careful design to avoid performance bottlenecks.

4. **Streamlit Usability**:

   Streamlit's user-friendly interface enabled expedited dashboard development; however, rendering extensive DataFrames resulted in minor performance delays. Optimizing data display (e.g., limiting row outputs) could improve user experience.

5. **Pipeline Modularity**:

   Organizing the pipeline into distinct phases enhanced maintainability and debugging.

6. **Scalability Considerations**:

   Although the current pipeline is adequate for small datasets, scaling to larger datasets will require replacing TinyDB with a more robust NoSQL solution (e.g., MongoDB) and optimizing PySpark for distributed clusters.

7. **Error Handling**:

   Implementing robust error handling (e.g., verifying CSV existence) was vital to prevent pipeline failures. Future iterations should incorporate logging and user-friendly error messages.